A MODEL FOR MPS PROCESSES AND ENVIRONMENTS

22 JUN 72

MPS 4.0

James G. Mitchell*

Xerox Palo Alto Research Center*
3180 Porter Drive
Palo Alto, CA 94304
(415) 493-1600

Stanford Research Institute
333 Ravenswood Avenue
Menlo Park, CA 94025
(415) 326-6200

This memo attempts to formalize the notions of process, process control states, inter-process control transfers, context, and naming environments for processes.

PROCESSES:

(Processes) A process has the following attributes:

(Control)

Control(S) is a pair (pc, status) consisting of a control pointer into some body of code associated with the process and a value denoting the status of S, chosen from the list in the branch labelled States below.

(Context)

Conceptually, the context for a process is the set of objects which the program can access by simple names. Since we view an activation record for a procedure, for instance, as a compound object whose components correspond to the local variables of the procedure, it is convenient to view the context of a process S simply as a vector of "references" to objects whose components can be accessed by simple identifiers in the source program.

An element of the context is a pair (CA, IND), where CA is the address of an object whose semantics matches S's requirements for the object specified by the context slot, and IND, if one, implies indirection (take the value of the object to which CA points as the CA for this entry).

The only way a process can touch any object is via the context vector. This includes the data objects called ports which are used for all control transfers, and the context vector itself (which must be accessed as a data structure for replacing context entries, for instance).

Accessing an object via a context entry whose CA value is NIL is not currently defined, but it would be nice if it would generate a signal.

(Ports) A port is simply a plug and a socket for forming a control connection from the port's process to another. A port has no state in its own right. The attributes of a port are the following:

(Owner) We denote the owning process of a port Q by Owner(Q).

(To) If a port Q is not connected, we say To(Q)=Nil; if Q is connected to another port Q', we say To(Q)=Q'.

Note that there is no requirement that To(To(Q))=Q.

Of course, To(Q)=Q is perfectly valid.

Sone more global definitions:

A configuration is just a set of processes.

We would like to arrange things so that a well-behaved configuration can have its ports interconnected and its processes started in any order.

CONTEXT OF A PROCESS:

It is NOT assumed that the context vector is physically attached to the data structure which contains the variables for the process.

There are a number of distinguished entries in every process's context (entries marked with a * are considered dynamic and must be set whenever a new incarnation of a process is created):

(SYSTEM) system transfer structure for access to system facilities; this is a component of every process's context, although it does not have to have the same value in them all.

(RETURN)* Pointer to port over which control will leave if S RETURNs.

(PENDING)* If S is Pending(Q), then the PENDING entry points to Q.

Initially the PENDING entry will point to a port "declared" at compile time called the process's RETURN port; the process's control pointer is initialized from information obtained at compile time also.

(CATCH)* The innermost catch phrase to be called if a signal is passed to S.

(SIGPATH)* pointer to the process to which signals which are not caught by S should go.

In the following list of allowable operations on context vectors, Ctx stands for a pointer to a context vector, i for an integer value, and x for an arbitrary value.

NewCtx ← CopyContext(Ctx);

SetContextEntry(Ctx, i, x);

x ← ReadContextEntry(Ctx, i);

DeleteContextEntry(Ctx, i);

Set the i'th context entry to NIL.

DeleteContext(Ctx);

PROCESS CONTROL STATES:

(States) The possible states of a process are:

(P) Pending(Q): Pending on port Q; i.e. control last left by a successful call through port Q.

This includes the case of one process starting another, which is just a call on a system facility (over a port of course)

When a process is created, it is initially in state P(START) where START is a distinguished port used as the inport for a function or the starting point for a process.

(R) Running.

At most one process can be in state R at a time.

(RESUMABLE)

Process can be started by control over any one of its ports or by a START operation directed at the process.

(Transitions) The transitions between the possible states of a process are represented in the following diagram:

FROM\TO:	Pending(Q)	Running	Resumable
P(Q):	NULL	control entry on Q	Invalid
R:	port call on Q	NULL	Signal generation
RESUMABLE:	port call on Q	START(process)	NULL

INTER-PROCESS CONTROL TRANSFERS:

Port Calls: the following MPS procedure describes port calls:

```
(PortCall) PROCEDURE(port, outparamlist);
   IF port.Owner # S THEN ERROR(InvalidPortCall, port);
   MakePending(S, port);
   (CheckFaults) DO BEGIN  ! loop until no problems with
   the control transfer
      (ForRetry) BEGIN
         IF (ObjectPort ← port.To) = NIL
            THEN BEGIN
               signal ← ResolutionFault;
               EXIT ForRetry;
            END;
         ResolvePort(ObjectPort, port);  !note that
         PortCall does this and not xfer.
         ObjectProcess ← ObjectPort.Owner;
         IF NOT Pending(ObjectProcess, ObjectPort)
            THEN BEGIN
               signal ← ControlFault;
               EXIT ForRetry;
            END;
         EXIT CheckFaults;
      END ForRetry;
      ! generate signal and anticipate control resumption
      via RESUME or port
         inparamlist ← SIGNAL(signal, port);
         IF outparamlist = NIL THEN RETURN (inparamlist);
   END CheckFaults;
   inparamlist ← xfer(port, ObjectPort, outparamlist);
   !basic control transfer
   RETURN (inparamlist);
END. PortCall
```

Note:

If a port is connected to itself, then its owning process immediately regains control as if the port call had not occurred at all.

The mechanism works correctly after any linkage fault is generated whether control arrives over the port or as the result of a RESUME by someone who caught the signal.

Procedure Calls: the following procedure describes procedure calls:

```
(ProcedureCall) PROCEDURE(port, outparamlist);

   NewProcess ← CopyProcess(port.To.Owner);
```

```
    inparamlist ← PortCall( Port(Owner: NewProcess, To:
    port.To), outparamlist); !now perform a normal port call

    RETURN (inparamlist);

END.
```

This description of the procedure call mechanism has a number of consequences:

The caller is specifying that a procedure call is to be made rather than the callee or the callee's inport specifying it.

The call is a two step operation involving the construction of a subsidiary port over which control goes after a copy of the callee is made. If this new port is not constructed, then the next time the caller uses the given port, it will no longer have Owner pointing to the protoprocess and the copy of the non-protoprocess may have altered lots of context entries.

The callee creates his local variables and enters them into his context himself; this is not done for him. It is assumed that the initial control pointer points at a place in his code body which will make an activation record for local values (this closely models procedures in most current Algol-like languages).

Possible solutions:

Let the inport to the callee contain the knowledge that it specifies whether a new copy of the process named by port.To.Owner is to be made. Then simple port calls would look exactly like procedure calls on the calling side. It also could allow the implementation of FORTRAN-like procedures which conceptually acquire local storage the first time they are called and then retain it thereafter.

This model of entry on a port is close to that proposed by BWL and suggests that the "knowledge" in the inport could simply be the address of some system facility for copying the procedure and pointing the procedure's RETURN port (which is copied as a consequence of copying the process ??) back at the caller's port. Note that a RETURN operation from the callee should not resolve the caller's port to the callee's RETURN port since that causes the problem that the caller does not want to go to the callee copy which returned to him, but to a new copy.

Signal Control:

Normally the SIGPATH context entry is altered in conjunction with the RETURN entry. When a signal is generated by a process, the innermost CATCH "procedure" is called with a local environment containing

(a) the signal code

(b) the paramlist which accompanies the signal code

The context within which the catch phrase is executed includes the part of the context of the process in which the catch phrase lives which is accessible to it.

A catch phrase may do one of two things which affect the signal propagation:

It may allow the signal to continue propagating. possibly stating the direction which it is to take (SIGPATH for the process containing the catch phrase defines the default direction).

It may do a "non-local" transfer of control into the body of its containing process S via the port on which S is pending. Prior to the actual resumption of S, another signal is passed from the point of generation of the original signal. This new signal, called UNWIND, destroys any processes which allow it to propagate. Once it reaches S, the resumption takes place.

During the time it is deciding which of these two courses to take, the body of a catch phrase may do any call or other evaluation which it pleases. However, all "backward" control transfers (RETURN, SIGNAL, ERROR, and EXITs which are not local to the body of the catch phrase) are interpreted as performed on behalf of S.

PROCEDURES AND PROCESSES AS DIFFERENT MANIFESTATIONS OF THE SAME PHENOMENON

This section explores the similarities between processes and procedures (in the traditional sense).

When a procedure is called in Algol the following events take place:

the caller constructs a parameter list

return linkage information is allocated in a place accessible to both the caller and the callee

the caller fills in the return information control passes to the entry point for the procedure in some body of code

the callee allocates space for local variables

when the callee is done, he deallocates the local variables

control passes back to the caller via the return link information

the parameter list is deallocated along with the return linkage information

In terms of our model for processes this paradigm can be restated as

the caller constructs a parameter record

a copy of the callee protoprocess is created: this includes his context information, and control/status

the callee's RETURN port is resolved back to the port which S is using for the "call"

the callee's context is altered to include the parameter record

control passes to the callee

the callee creates an instance of its activation record

when the callee is done, he allocates and constructs a return record

the callee frees his activation record

control passes back to the caller over the process's RETURN port and the callee copy is destroyed

PROCESS CREATION:

An instance of a process is nothing more than a (Control, Context) pair. Processes can be created by copying an already existing process (however, this is not quite what one would like, namely copies of the data structures created by the process itself -- but see the next paragraph). Initially a process is created from some virgin form which has usually been established from a file. We will call such an object a protoprocess: it is not an executable entity, but holds a place in the naming environment and creating a process from it is a simple operation.

A protoprocess consists of a partially initialized context and initial control information. If the records created by the process for local variables, etc. could be created independently of one another, then making a copy of an already existing process and the data structures owned by it would be a simple operation. In general this is not the case: records contain references to other records, and hence, truly copying a process is equivalent to copying a set of inter-referential records. I don't think we should provide a built in facility to do this -- it is a job for someone using the system.

Creating a new process S from some already existing process or protoprocess P is simply a matter of copying the control and context information for P to S.

PROCESS NAMING ENVIRONMENTS:

Compile Time:

The local variables for a process or procedure are those declared following the header statement for the process.

The following example demonstrates this:

```
(Ex1) PROGRAM (a1, b1);
   DECLARE r1, s1, t1;
   body-1
   (Ex2) PROGRAM (a2, b2);
      DECLARE r2, s2, t2;
      body-2
      END.
   END.
```

When an incarnation of Ex1 is initially created. space is allocated for a1, b1, c1, r1, s1, and t1. Thereafter whenever Ex2 is called (which is equivalent to creation followed immediately by control transfer), a2, b2, ..., t2 are allocated and will be deallocated only when Ex2 does a RETURN.

The prototype program from which a process can be created is the following:

```
(Example) PROGRAM (parameter-list);

   local-variable-declarations;

   program-body

   END.
```

Any gathering of many program prototypes in one source file is simply a way of binding some contexts before process creation time and of causing one CREATE operation to result in the creation of a number of processes. Stated differently, a source module is a means of binding processes into configurations before creation time.

A local-variable-declaration may be

a program declaration: this allows Algol-like bindings of context.

an INCLUDE declaration: incarnations of any objects declared in the INCLUDE module will have the same lifetime as normal local variables.

Execution Time:

The execution time naming environment consists of a tree whose nodes are processes and instantiations of data modules. More than one instance of a process or data module can reside at a node of the tree. Also, separate instances of the same process may reside at different nodes in the naming tree. A given process resides at exactly one node in the naming tree.

The naming environment is not necessarily coupled with the control or context of processes although it is often convenient for them to be associated. All normal bindings of names to objects use the compile time symbol table associated with a process as the most local information, and the naming tree as the next source of names.

We add the following attributes to those listed above for processes:

(Parent) Parent(S) is S's ancestor in the naming tree.

(Sibling) Sibling(S) is a process such that Parent(S)=Parent(Sibling(S)) or Sibling(S)=NIL

(Child) Child(S) is the "first" descendant process of S in the naming tree. The children of a process are well-ordered, and the following loop will access all the immediate descendants of S:

```
child ← Child(S);
UNTIL child=NIL
   DO BEGIN
      process this child;
      child ← Sibling(child);
   END;
```

EXAMPLE:

Primitive Operations for Process Creation and Calling Procedures:

```
(CreateFromFile) PROCEDURE(filename);

   DECLARE POINTER(PORT) CallInPort;

   a ← MapIn(filename);  ! map file into addressable memory

   CallInPort ← ProtoProcess(a.InitialPC, a,
   self.ReturnPort.To); !make a protoprocess with initial
   control from the file and parent my caller

   RETURN(CallInPort);  ! give back address of port by
   which process can be called

END.

(ProtoProcess) PROCEDURE(pc, codebase, parent);
!make a protoprocess with initial pc as given in the
codebase given and with the specified parent process

   DECLARE POINTER(PROCESS) p;

   p ← CopyProcess(SkeletonProcess, parent);  ! make a
   minimal, virgin process

   p.Control ← pc;

   p.Context.Pending ← S(p.ReturnPort);  ! initial state is
   Pending(ReturnPort)

   p.Context.CodeBase ← codebase;

   StartUp.To ← p.ReturnPort;

   xfer(StartUp, StartUp, NIL);

   RETURN(StartUp.To);  ! really not necessary since
   StartUp belongs to caller of ProtoProcess

END.
```

Sample Program Outline:

```
(a) ROUTINE (pa, qa);

   DECLARE xa, ya, za;

   (a1) ROUTINE (pa1, qa1);

      DECLARE xa1, ya1, za1;
```

```
         body of a1;

      END of a1.

      body of a;

   END of a.
```

The following purports to be the code generated by the MPL compiler for the sample program above:

```
   Proto-code for a's protoprocess

      (aProto)

      DECLARE PROCESS p, POINTER(PROCESS) ap, POINTER(PORT)
      caller;

      p ← CopyProcess(SkeletonProcess, self);  !prototype
      process descriptor for a

      ! set any of p's context which is desired

      p.Control ← $aBEGINS;

      xfer(ReturnPort, ReturnPort, self.inargs);

      ! The following loop handles creation and calling of
      instances of a.

         DO BEGIN  ! loop forever

            ap ← CopyProcess(p, self);  !copy of preset
            process descriptor for a

            caller ← ReturnPort.To;

            ReturnPort.To ← ap.ReturnPort; !so can transfer
            control to ap and leave aProto pending ReturnPort.

            xfer(ReturnPort, caller, self.inargs);

            ! aProto is left pending his ReturnPort and has
            cut himself out of the control path from the
            caller to the instance of a.

         END;

   Code for the routine a:

      (aBEGINS)

      DECLARE xa, ya, za, PORT CALLa1;
   CALLal.Owner ← self;

   CALLal.To ← ProtoProcess(alProto, alProto, self);

   body of a

code for al's protoprocess

   (alProto)

   DECLARE PROCESS p, POINTER(PROCESS) alp, POINTER(PORT)
   caller;

   p ← CopyProcess(SkeletonProcess, self);

   p.Context.GLOBALS ← ReturnPort.To.Owner.Context.LOCALS;
   !local variables of enclosing preear included in context
   of any incarnation of al.

   p.Control ← $alBEGINS;

   xfer(ReturnPort, ReturnPort, self.inargs);

   ! The following loop handles creation and calling of
   instances of a.

      DO BEGIN  ! loop forever

         alp ← CopyProcess(p, self);  !copy of preset
         process descriptor for a

         caller ← ReturnPort.To;

         ReturnPort.To ← alp.ReturnPort; !so can transfer
         control to alp and leave aProto pending
         ReturnPort.

         xfer(ReturnPort, caller, self.inargs);

         ! alProto is left pending his ReturnPort and has
         cut himself out of the control path from the
         caller to the instance of al.

      END;

code for al

(alBEGINS) ! code for al

DECLARE xal, yal, zal;  ! make local record for self.

body of al
```